A DP Approach to Hamiltonian Path Problem

Dmitriy Nuriyev

January 16, 2013

1 Abstract

A Dynamic Programming based polynomial worst case time and space algorithm is described for computing Hamiltonian Path of a directed graph. Complexity constructive proofs along with a tested C++ implementation are provided as well. The result is obtained via the use of original colored hypergraph structures in order to maintain and update the necessary DP states.

2 Introduction

Dynamic Programming hardly requires introduction. Since the term was in introduced by Richard Bellman in 1940's it has countless applications and it's power allows to have fast algorithms in cases where at a blush there seems to be no way to avoid exponential time. Such examples include graph algorithms: Bellman-Ford, Floyd-Warshall as well as option pricing, numerical solutions to HJB equations ("backwards in time"), discrete optimal control policies, knapsack small block size problem [2], Smith-Waterman local sequence alignment algorithm and many others.

Hamiltonian path (H-path) in directed and undirected graph is one of Karp's 21 NP-complete problems [3]. The question it asks is to find cycle or path in a given graph which visits every vertex exactly once. Therefore one can see that H-path is the shortest path which visits all nodes in the graph and thus provides the answer to the transportation problem in question.

David Zuckerman [4] showed in 1996 that every one of these 21 problems has a constrained optimization version that is impossible to approximate within any constant factor unless P = NP, by showing that Karp's approach to reduction generalizes to a specific type of approximability reduction. Significant progress has been made in expanding the class of graphs for which polynomial solution does exist: notably Ashay Dharwadker [5] discovered an algorithm for a broad class of highly connected graphs.

The intuition behind the presented approach comes from physical objects that seem to have a way to solve some NP-complete problems such as soap bubbles forming (almost) minimal surfaces and protein folding satisfying hydrophilic-hydrophobic Boolean conditions.

Consider a cobweb attached to a tree with one end and being pulled by the other end. One can observe that the lowest (average) strain or highest slack in the cobweb is achieved along the longest sequence of web segments. This is of course a Hamiltonian path in the graph represented by the cobweb. If one wants to find it then it makes sense to implement some incremental scheme which akin to a difference scheme for PDE with boundary condition, computes the strains incrementally starting with one attachment point and moving towards the opposite attachment point.

3 Definitions

Denote $G, V(G), E(G)$ - the input directed graph, it's vertices (aka nodes) and edges respectively, $n = |V(G)|$. Assume G to be a general directed graph which does not have multiple edges between any two vertices and that there are no loop edges.

A given instance of H-path problem is represented by a triple (G, s, e) where $s, e \in V(G)$ and the task is to find path $\pi(s, e) \in V(G)$ starting at node s and ending at e so that every vertex in $V(G)$ is visited and only once.

In the following description I shall opt for a less conventional but more compact and practical terminology in describing the main algorithm by adopting Object-Oriented terminology. The entities used here will be Objects which consist of Attributes - representing data and Methods - representing functions as well as traditional program functions and variables - these are distinct from object methods and attributes in that they are "static", effectively having global scope and accessible from all objects.

If an object A is an attribute of B I typically write A_B unless making a statement about any object A or it is clear from the context what B is. Also instead of single letter notation I elect mnemonics that are much shorter than full object names but are easier to associate with full object names despite sounding weird.

I start with introducing a concurrent graph traversal method which unlike DFS/BFS (Depth First Search/Breadth First Search) has some useful properties in this context. This will become a basis for the workflow of main algorithm.

Denote $P(G)$ family of H-paths in G between nodes s, e.

Definition 3.1. Let *botmaster* (C++ code: edg) be an object with the following:

- Attributes:
  (1) *bots* - family of all *bot* objects, defined below.
  (2) march step $marchStep \in 1..n$
  (3) *dock* a Boolean variable associated with pair $(v, marchStep)$, $v \in V(G)$
- Methods:
  (1) *march* for all $bot \in bots$ invokes $advance_{bot}$, defined below.
  (2) $addDelete(v_1, v_2)$ (C++: edg::add_delete) defined below

Definition 3.2. Let *bot* be an object invoked by master object *botmaster*, which on march step *marchStep* conditionally transitions from node a to its neighbor b along edge ab, denote that fact as $bot(a, s) \to b$. Bot's only attribute is current node $a \in V(G)$ and it has

- Methods:
  i *advance*:
    (1) $bot(a, s) \to b \Rightarrow dock(b, s) = F$, $dock(a, s-1) = T$. This means dock acts like mutex for bots - first bot accessing b docks at b.
    (2) $addDelete(a, b)_{botmaster}$ is invoked
    (3) If $bot(a, s) \to b_1 ... b_h, h > 1$, $bot(a, s)$ spawns botlets $bot(b_1, s+1) ... bot(b_h, s+1)$
  ii *terminate* - deletes *bot* object under the following conditions:
    (1) If $\nexists b : bot(a, s) \to b$, $bot(a, s)$ is terminated.
    (2) $bot(a, n-2) \to e$ and there are no other $v : bot(a, n-2) \to v$ bot march stops.

Definition 3.3. Let $hist(bot)$ be the history of nodes visited by *bot*, this includes "genetic memory", i.e. newborn botlet inherits history of it's parent.

Remark 3.1. There are no more than n bots at any given time.

This follows from correspondence between bots and nodes guaranteed by dock condition (3.2).

Remark 3.2. Let π is H-path on G. Then there exists a *bot*: $hist(bot) = \pi$.

Proof. Assume that $\pi_1 = (s, ..., e_1) \subset \pi$, is the longest path traversed by any bot in $|\pi_1|$ steps. By (3.2).1 we have a bot docked at e_1 which will traverse to $e_2 : (e_1, e_2) \subset \pi$ on step $|\pi_1| + 1$ thus violating the assumption of maximality of π_1. □

Remark 3.3. Any vertex of G is visited at most n times.

This follows from condition (3.2).ii.(2)

Definition 3.4. Let

- Color C of graph g for node $v \in G$ be an integer corresponding to a class of paths $\{(s...v)\}$ in graph g.
- $colors(g)$ be all colors of graph g. Also:
- Each color has a unique node it relates to via surjective function: $cono : cono(C) \to V(g)$ (stands for color nodes: C++ dg::color_nodes).
- Denote $\pi(C)$ set class of paths for color C. Note that $|\pi(C)|$ may be exponentially big and therefore this is not explicitly computed by the algorithm.

- There is a base color which s is painted in - *emptycolor*

Definition 3.5. Let,

- *cohi* (color hierarchy, C++ color_hierarchy) be a directed graph of colors, where edge c_1c_2 denotes that every path over nodes of g in c_1 is a subpath of some g-path in c_2, has form $\{(c, S_c), S_c \subset colors(g)\}$
- $colors(cohi) \equiv V(cohi)$ returns colors of *cohi*
- $suco(C)$ is a global function (C++ sub_colors) which returns all colors in *cohi* which have path to C, i.e sub-colors of C, also has form

$$\{(c, S_c), S_c \subset \bigcup_{v \in V(G), slack=1..n-1} colors(g_{pagra(v,slack)})\}$$

- $noco(v)$ (node colors, C++: dg::node_colors) is colors over node $v \in V(g)$, has form $\{(v, S_v), S_v \subset colors(g)\}$
- $cn(v)$ (colors for node, C++: dg::cn) is all colors C over v such that $cono(C) = v$
- $top(cohi), base(cohi)$ are respectively first (no ancestors) and last (no descendants) colors in *cohi* graph.

Lemma 3.1. $\forall v \in g, |cn(v)| = O(n)$

Proof. Be remark (3.3) we have $O(n)$ bit visits to v. Each visit performs merge for given $pagra(v, slack)$ which generates new $C : cono(C) = v$. □

Definition 3.6. Color C is called inactive: $inactive(C)$ iff

- $cono(C) = \emptyset$
- or $cono(C) \notin V(g)$
- or $noco(cono(C)) = \emptyset$
- or $C \notin top(cohi) \wedge cohi(C) = \emptyset$

Definition 3.7. Let $S \in colors(cohi), a \in V(g)$ $dep(S|cn(a))$ is a subset of S such that either

$$C \in S, \nexists \pi(C, top(cohi)) \in cohi : cn(a) \cap \pi(C, top(cohi)) = \emptyset$$

Or

$$C \in S, \nexists \pi(C, base(cohi)) \in cohi : cn(a) \cap \pi(C, base(cohi)) = \emptyset$$

Definition 3.8. For all $a \in V(G), slack \in 1...n$ call path graph $pagra(a, slack)$ (C++: dg) an object which includes all paths $(s...a) \subset G$ of length *slack* and has:

- Attributes:
  (1) Directed graph g with nodes in $V(g)$
  (2) Node colors $noco(V(g)) = \{(n_k \to \{c_k^1, .., c_k^h\})\}$, *noco* is a one-to-many map $V \to C$
  (3) Color nodes $cono(C)$
  (4) Color hierarchy $cohi(g)$
- Methods:
  (1) $addSlack(b), b \notin V(pagra(a, slack))$, does:
    i adds edge ab to $g_s, \forall m$
    ii invokes *paint* which performs:
      (1) allocates new C
      (2) $\forall v \in V(g), noco(v) := (noco(v), C)$
      (3) $cono(C) := b$
      (4) $cohi(C) := top(cohi)$

(4) $suco(C) := colors(cohi)$

(2) Remove node (C++: dg::rm_node) $reno(a)$:

i deletes $noco(a)$

ii executes $bleach(cn[a])$ which:

(1) computes $D = \{dep(cn[v]|cn[a])|\forall v \in V(g)\}$

(2) $\forall C \in D$ remove C from all attributes of $pagra(a, slack)$

(3) ensures $\forall C \in colors(cohi), inactive(C) = F$

(4) every color $C \in cohi$ has a direct predecessor or $bleach$ removes such C

(5) if $noco(v) = \emptyset$, remove v from g $\forall v \in V(g)$

(3) Merge $pagra(a) + pagra(b)$: unions are taken for $cohi, cono, noco$, i.e. it has form:

$$\{(a, S_a)\} + \{(b, S_b)\} = \{a, b : a = b \Rightarrow (a, S_a \cup S_b), otherwise(a, S_a), (b, S_b)\}$$

Definition 3.9. Denote

$$sucoDFS(a, X, suco, dir), a \in colors(cohi), X \subset colors(cohi), dir \in \{up, down\}$$

a *pagra* method (C++: dg::has_path_to_top()) which traverses *cohi* starting from color a, bypassing colors X (C++: dg::xcolors) and uses *suco* to make transitions between nodes as follows:

(1) $dir = up$

(1) run regular DFS against graph *cohi* starting at color a

(2) when next node is $c \in X$ return to the next available sibling node

(3) transition from color vertex a_1 to a_2 only if $\pi_1 \subset suco(a_2)$ where $\pi_1 \equiv (a...a_1)$ is a sequence of colors - path - in *cohi* traversed before a_2.

(4) terminate and return path π when next node is in $top(cohi)$

(5) when all colors have been traversed terminate and return $\emptyset$

(2) $dir = down$

(1) run regular DFS against graph $inv(cohi)$ starting at color a

(2) when next node is $c \in X$ return to the next available sibling node

(3) transition from color vertex a_1 to a_2 only if $\forall c \in \pi_1 a_2 \in suco(v)$

(4) terminate and return path π when next node is in $base(cohi)$

(5) when all colors have been traversed terminate and return $\emptyset$

Remark 3.4. By definition of $sucoDFS$ we have:

$$sucoDFS(a, X, suco, up) = \emptyset \vee sucoDFS(a, X, suco, down) = \emptyset \Rightarrow dep(a|X) = \{a\}$$

Definition 3.10. Given directed graph G $inv(G)$ inverts arc orientation.

Definition 3.11. Slacks $slacks(a)$ is an object, effectively a wrapper for $pagra(a, slack)$ with:

- Attributes:

(1) A single node $v \in V(G)$

(2) A set of slacks S

(3) Path graphs $pagra(a, slack), slack \in S$

- Methods

(1) Add slack - $addSlack(b), b \notin V(g_{pagra(a,slack)})$, invokes $addSlack(b)_{pagra(b,slack)}, \forall slack$

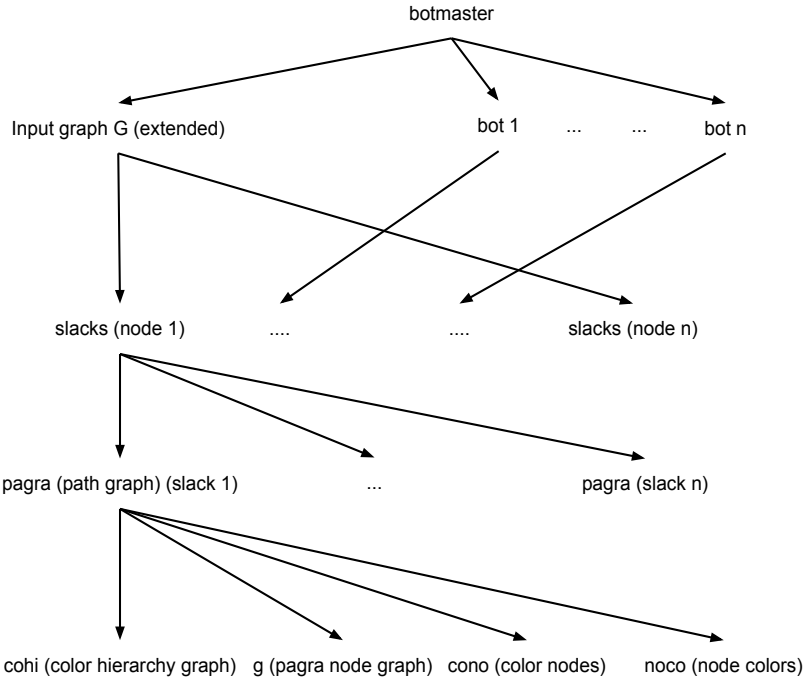
(2) Remove node: $reno(a)$ invokes $reno(a)_{pagra(b,slack)}, \forall slack$

(3) Merge $slacks(a) + slacks(b)$: invokes $pagra(a, slack) + pagra(b, slack), \forall slack \in S$

Definition 3.12. $addDelete(a, b)_{botmaster}$ is a method which performs the following steps:

- $reno_{slacks(a)}(b)$
- $addSlack_{slacks(a)}(b)$
- Merge $slacks$: $slacks(b) := slacks(a) + slacks(b)$

Figure 1: Main objects



Definition 3.13. For a connected sequence of nodes π $dup(\pi)$ is a set of repeated nodes.

Definition 3.14. Denote $P'(G)$ class $sucoDFS$-traversible paths $\forall s, e \in V(cohi), X = \emptyset$.

4 Explanation

The idea here is to categorize families of paths without explicitly describing each one of them. *Slacks* object for a given node encodes paths to a given node over G and groups them by length (slack). For each given slack *pagra* object is the encoding of paths given by all paths traversible in it's own copy of graph g. Graph g by itself is not enough to keep track of those paths due to the way it has to be updated: via *merge* and *reno* operation. *reno* operation ensures paths don't run over the same node twice since *addDelete* removes each node v from g when $bot(u, marchStep) \to v$. After that, *merge* executes $pagra(u, slack) + pagra(v, slack)$ (if $pagra(v, slack) \neq \emptyset$). If $g_1 = g_{pagra(u,slack)}$ and $g_2 = g_{pagra(v,slack)}$ both contain some node w then the resulting g will lose track of paths through w resulting in "synthetic path" which is not a path in G for a given *slack*. To preserve actual paths, concept of color is introduced. However just the color alone is not enough since we have cases when we start with some color, for example, "red" in $slack_0$ which then "splits" into $slack_1$ and $slack_2$ each one carrying it's own version of "red" which then gets modified by their own *reno* operations. After a while slacks combine again but now red two distinct path classes. We avoid exponential explosion in number of colors by using *cohi*, which associates each copy of *red* with some own unique color added in $slacks_1$, for example, "blue" and in $slacks_2$ - "brown". These new colors are added as part of *addDelete* method in bot march and so their number is "small" as shown below. When $slack_1 + slack_2$ is computed, *suco* includes $(brown, (red, ...))$, $(blue, (red, ...))$. Method $sudoDFS$ uses *suco* to avoid "synthetic path" as shown below.

5 Path existence

Lemma 5.1. *Total number of colors is* $|colors(suco)| = O(n^3)$.

Proof. Indeed, a color is assigned by *paint* method which is called for:

- every vertex in $v \in V(G)$ - $O(n)$
- every bot visit to v - $O(n)$ by Remark (3.3)
- every $pagra(v, slack)$ - $O(n)$

□

Lemma 5.2. *Let* $\pi \in P'(cohi_1 + cohi_2)$ *then*

$$\nexists \pi_1 \subset P(cohi_1), \pi_2 \subset P'(cohi_2) : \pi = (\pi_1, \pi_2)$$

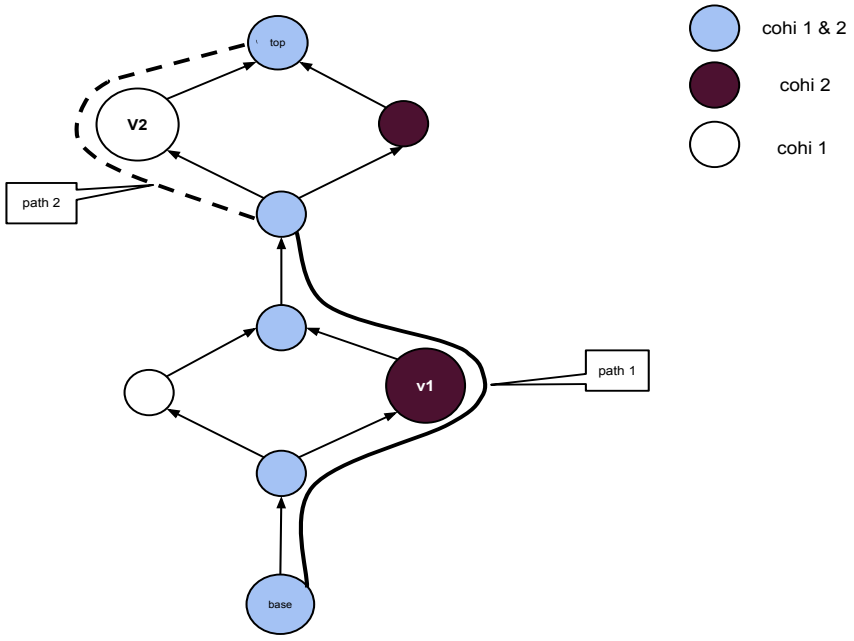
i.e. sucoDFS does not return π *that starts in* $cohi_1$ *and continue in* $cohi_2$.

Proof. Assume such path π existed, then we would have

$$c_1 \in colors(cohi_1) \backslash colors(cohi_2), c_2 \in colors(cohi_2) \backslash colors(cohi_1) : c_1, c_2 \in \pi$$

See Figure 2 for illustration. But then we must have $c_1 \in suco(c_2)$ by rules (3.7).1-2 which is impossible because c_1 is not even in $colors(cohi_2)$

Figure 2: Impossible synthetic path on merger



□

Lemma 5.3. *Let* γ *be a sucoDFS-traversible path,* $\pi = cono(\gamma), u \in \pi$ *and importantly* $v \in g$ *but* $v \notin \pi$, *then*

$$dep(cn(u)|cn(v)) = \emptyset, \forall u \in \pi$$

Proof. By definition of *dep*, $\forall u \in \pi$, $cn(u)$ has path to $top(cohi)$ and that path is γ and $\gamma \cap cn(v) = \emptyset$ therefore $dep(cn(u)|cn(v)) = \emptyset$. □

Lemma 5.4. *Let*

$$\pi = cono(sucoDFS(top(cohi_{pagra(b,j)}), \emptyset, suco, down))$$

then $dup(\pi) = \emptyset, |\pi| = j$.

Proof. Assume there was an intersection $dup(\pi) \neq \emptyset$ then we have $c_1, c_2 : cono(c_1) = cono(c_2)$ however if c_2 was traversed after c_1 then by (3.7).1 observe that $c_1 \in suco(c_2)$ but by *suco* construction this is impossible if $cono(c_1) = cono(c_2)$.

$|\pi| = j$ folows from *bleach* method definition: every color $C \in cohi$ has a direct predecessor or *bleach* removes C, also *paint* method and *cono* surjectivity ensures that any color path in $cohi_{pagra(b,j)}$ from *base* to *top* is j long. □

Lemma (5.4) allows us to recover final H-path from $cohi_{pagra(e,n-1)}$ using standard $sucoDFS$ function. That is if such path is included in $cohi_{pagra(e,n-1)}$, let's show this is indeed the case:

Theorem 5.5. *If* $P(G) \neq \emptyset$ *then* $\exists \pi \in P(G)$*:*

$$\pi = cono(sucoDFS(top(cohi_{pagra(e,n-1)}), \emptyset, suco, down))$$

Proof. Using induction by march step, this trivially holds at node s, $marchStep = 1$, assume at $marchStep = k$ we have:

$$\gamma_j \equiv sucoDFS(top(cohi_{pagra(v_j,j)}, \emptyset, suco, down)$$

Let $\pi_j \equiv cono(\gamma_j)$. When $bot(v_j, j) \to v_{j+1}$ we can show that, for $\pi_{j+1} \equiv (\pi_j, v_{j+1})$

$$\pi_{j+1} = cono(sucoDFS(top(cohi_{pagra(v_{j+1},j+1)}, \emptyset, suco, down))$$

Indeed $bot(v_j, j) \to v_{j+1}$ invokes *addDelete* which invokes methods *merge* and *reno*, *merge* does not affect π_j by lemma (5.2) while if $v_{j+1} \in V(g_{pagra(v_j,j)})$ *reno* will invoke $bleach(dep(cn(v)|cn(v_{j+1}))), \forall v \in \pi_j$ but since π_j is $sucoDFS$ traversable and $v_{j+1} \notin \pi_j$ therefore by lemma (5.3):

$$colors(\gamma_j) \not\subset dep(cn(v)|cn(v_{j+1}), \forall v \in \pi_j$$

therefore γ_j and therefore π_j is not affected by $bot(v_j, j) \to v_{j+1}$. □

6 Algorithm

- Start with one bot which current node set to s
- *slacks* object and the associated *pagra* objects are initialized
  - *slacks* consists of just one *pagra*
  - g_{pagra} consists of only one node s
  - *cohi* consists of just *emptycolor*
  - *noco* is just $(s, emptycolor)$ pair
  - *cono* is $(emptycolor, s)$
- Bot advances $s \to v_1$ according to it's rules, and invokes *addDelete* function with updates $slacks(v_1)$
- Steps above are repeated, until $pagra(e, n-1) \in slacks(e)$, i.e. we have a path graph with slack equal to $|V(G)| - 1$.
- Extract the H-path by computing

$$cono(sucoDFS(top(cohi_{pagra(e,n-1)}), \emptyset, suco, down))$$

7 Complexity

Time complexity is based on that:

- there are $O(n)$ G nodes
- visited by bots at most $O(n)$ times
- each visit requires *runo* call for each of $O(n)$ graphs $pagra(v, slack)$
- *reno*, requires $sucoDFS$ call for each color a of at most $O(n)$ sized (lemma (3.1)) $cn(v)$ color set
- $sucoDFS(a, cn(b), suco, dir)$ complexity is $O(n^3)$ since:
  - any $sucoDFS$-traversible path in *cohi*: $\pi(base(cohi), top(cohi))$ is $O(n)$ long by lemma (5.4)
  - set $X = cn(b)$ cardinality is at most $O(n)$ by lemma (3.1) and therefore DFS requires at most $O(n)$ rollbacks when X color is encountered
  - validating conditions (3.7).1,2 on every step costs $O(n)$
- which is executed for each $O(n)$ nodes $v \in g$ to decide if the node is bleached

Space complexity is based on that:

- there are $O(n)$ G nodes
- each node has $O(n)$ *pagra* objects
- each *pagra* object has *cohi* as biggest attribute
- each *cohi* attribute has at most $O(n^3)$ colors

Thus total current worst case time complexity is $O(n^8)$ and space complexity $O(n^5)$. Average case complexity is typically much lower because in practice $cn(v)$ is $O(1)$ sized and average number of node bot revists is also $O(1)$.

$sucoDFS(a, cn(b), suco, dir)$ complexity can be lowered by $O(n)$ if we eliminate $cn(b)$ from *cohi* first which is done once for all colors a. Than for each a we only have to run $sucoDFS(a, \emptyset, suco, dir)$ in $O(n)$.

Complexity can likely be further lowered by an additional $O(n)$ by consolidating all *cohi* per G node and across *pagra* of different slack levels. Slack values can be extracted from *cohi* directly when necessary.

Additionally $dep(cn(b)|cn(a))$ can possibly be found faster if we start with an encoding of all $sucoDFS$-traversible paths P to top. Then on every update of *cohi* instead or rerunning $sucoDFS$ we could just update P.

8 Implementation and testing

The algorithm was tested on 10000 randomly generated graphs each with 17 nodes and 3 outbound edges per node. Test graphs were generated using function $dg :: gen\_graph$ which is part of the source code and is as follows;

- Generate a random H-path first. Sample without replacement from a discrete uniform random distribution over $1...n : U(1...n)$.
- For each node on selected H-path generate given number of edges δ by choosing vertex 1 sequentially from the path and then choosing vertex 2 by sampling without replacement from $U(1...n)$ δ times.

Please note that performance tuning was not a priority therefore the author is aware of a number of inefficiencies.

The algorithm is available for download at
https://docs.google.com/folder/d/0B3s6PXhKJO6HWnE4c0VZbVJodDQ/edit .

It is free to use for academic and educational purposes use but requires a license for commercial and government use. Potential commercial and government users should note the algorithm has a Patent Pending status with USPTO.

References

[1] Garey, Michael R.; David S. Johnson (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman. ISBN 0-7167-1045-5

[2] S.Martello, D.Pisinger, P.Toth "Dynamic Programming and Strong Bounds for the 0-1 Knapsack Problem", submitted Management Science (1997)

[3] R. Karp "Reducibility Among Combinatorial Problems", Complexity Of Computer Computations. New York: Plenum pp 85-103 (1972)

[4] D. Zuckerman (1996). "On Unapproximable Versions of NP-Complete Problems". SIAM Journal on Computing 25 (6): 12931304. doi:10.1137/S0097539794266407.

[5] Ashay Dharwadker, "A New Algorithm For Finding Hamiltonian Circuits", 2004

9 Appendix

Figure 3: Source Code

C:\mydocs\deform_\sdg.h 1

```
#include "stdafx.h"
#include "util.h"

using namespace std;
// shared data:

int emptycolor=-100;
int add_cnt=1;
int new_color=1;
int start_node, end_node;
int gr_rm_node_cnt=0;
int add_delete_cnt=0;
int step;
const int max_colors=1000000;
int color_slacks[max_colors];
map2 subcolors;

struct dg : util {
    map2 *rmc;
    map1 nodes;
    int slack_level,top_node;
    map1 top_colors;
    map2 node_colors; // int value for each node from color enum above; used to do rm_node
    // when they want to see if undeground streams connect they release a dye and see where it comes out
    map2 color_hierarchy;
    map<int,int> color_nodes; // color to top node of graph which got painted in this color
    map2 bleach_nodes;
    dg() {alloc();}
    ~dg() { }
    void alloc() {
        rmc = new map2();
        top_colors[emptycolor]=true;
        slack_level=1;
        color_nodes[emptycolor]=start_node;
        node_colors[start_node][emptycolor]=true;
    }
    dg(const dg& b) {
        alloc();
        color_hierarchy=b.color_hierarchy;
        descendants=b.descendants;
        precendants=b.precendants;
        slack_level=b.slack_level;
        color_nodes=b.color_nodes;
        node_colors=b.node_colors;
        top_colors=b.top_colors;
        top_node=b.top_node;
        nodes=b.nodes;
    }
    void rmrefs(int rn,map2 &cendants) {
        map2::iterator mit;
        for (mit=cendants.begin(); mit!=cendants.end(); mit++) {
            int node=(*mit).first;
            if (cendants[node].count(rn)>0) {
                cendants[node].erase(rn);
            }
        }
    }
    void add(int to,int from){
        top_node=to;
        descendants[from][to]=true;
        precendants[to][from]=true;
        if (node_colors.count(top_node)==0) paint();
    }
    map1& get_nodes() {
        map2::iterator mit;
        nodes.clear();
```

```
        for (mit=descendants.begin(); mit!=descendants.end(); mit++) {
            int node=(*mit).first;
            nodes[node]=true;
        }
        for (mit=precendants.begin(); mit!=precendants.end(); mit++) {
            int node=(*mit).first;
            if (nodes.count(node)==0) {
                nodes[node]=true;
            }
        }
        return nodes;
    }
    map1 get_nodes(map2& desc) {
        map1 tn;
        map2::iterator mit;
        for (mit=desc.begin(); mit!=desc.end(); mit++) {
            int node=(*mit).first;
            tn[node]=true;
            map1::iterator it;
            for (it=desc[node].begin();it!=desc[node].end();it++) {
                int n=it->first;
                tn[n]=true;
            }
        }
        return tn;
    }
    int paint() {
        int ret=0;
        perm_paint(node_colors,new_color);
        color_hierarchy[new_color]=top_colors;
        color_nodes[new_color]=top_node;
        ret=new_color;
        color_slacks[new_color]=slack_level;
        top_colors.clear();
        top_colors[new_color]=true;
        subcolors[new_color]=flatten(color_hierarchy);
        new_color++;
        return ret;
    }
    void clear_inactive_colors() {
        list<int> rml;
        map2::iterator it;
        for (it=color_hierarchy.begin();it!=color_hierarchy.end();it++) {
            int c = (int)it->first;
            if (!active_color(c)) rml.push_back(c);
            map1::iterator jt;
            map1 x = color_hierarchy[c];
            for (jt=x.begin(); jt!=x.end();jt++) {
                int sc = (int)jt->first;
                if (!active_color(sc)) {
                    color_hierarchy[c].erase(sc);
                }
            }
        }
        list<int>::iterator lit;
        for (lit=rml.begin();lit!=rml.end(); lit++) {
            int c = (int)*lit;
            color_hierarchy.erase(c);
        }
    }
    dg& operator +=(dg& g2) {
        add_cnt++;
        merge_maps(color_hierarchy,g2.color_hierarchy);
        merge_maps(color_nodes,g2.color_nodes);
        merge_maps(descendants,g2.descendants);
        join_maps(&top_colors,&g2.top_colors);
```

```
        upd_precendants();
        merge_maps(node_colors,g2.node_colors);
        get_nodes();
        clear_inactive_colors();
        return *this;
    }
    void upd_precendants() { // from descendants
        map2::iterator it;
        map1 cd_desc;
        precendants.clear();
        for (it=descendants.begin();it!=descendants.end(); it++) {
            int node = it->first;
            map1 &descs = it->second;
            map1::iterator jt;
            for (jt=descs.begin(); jt!=descs.end(); jt++) {
                int node2=jt->first;
                precendants[node2][node]=true;
            }
        }
    }
    map2 ich;
    void inverse_ch() { // from clor_hierarchy
        ich.clear();
        map2::iterator it;
        map1 cd_desc;
        map1 emptymap;
        for (it=color_hierarchy.begin();it!=color_hierarchy.end(); it++) {
            int sup_color = it->first;
            if (ich.count(sup_color)==0) ich[sup_color]=emptymap;
            map1 &descs = it->second;
            map1::iterator jt;
            for (jt=descs.begin(); jt!=descs.end(); jt++) {
                int sub_color=jt->first;
                ich[sub_color][sup_color]=true;
            }
        }
    }
    map2 cn;
    void inverse_color_nodes() {
        map<int,int>::iterator it;
        for (it=color_nodes.begin();it!=color_nodes.end();it++) {
            int color = (int)it->first;
            if (!active_color(color)) continue;
            cn[color_nodes[color]][color]=true;
        }
    }
    void perm_paint(map2 &node_colors,int color){
        get_nodes();
        map1::iterator mit;
        int node=top_node;
        while (true) {
            if (node==start_node) break;
            if (precendants[node].size()>1) break;
            int i = precendants[node].size();
            node=(int)precendants[node].begin()->first;
        }
        for (mit=nodes.begin(); mit!=nodes.end(); mit++) {
            int node=(int)(*mit).first;
            node_colors[node][color]=true;
        }
    }
    void get_top_colors(int c,map1 &ret) {
        map<int,int>::iterator it;
        for (it=color_nodes.begin();it!=color_nodes.end();it++) {
            int color = (int)it->first;
            int node = (int)color_nodes[color];
```

```
            if (node==top_node) {
                ret[color]=true;
            }
        }
    }
    bool active_color(int c) {
        if (color_nodes.count(c)==0) return false;
        int cn=color_nodes[c];
        if (nodes.count(cn)==0) return false;
        if (node_colors.count(cn)==0) return false;
        if (node_colors[cn].count(c)==0) return false;
        return true;
    }
    map1 all_bleached_colors;
    void rm_node(int X,int Y) {
        gr_rm_node_cnt++;
        map1::iterator it;
        map1 emptymap;
        inverse_color_nodes();
        map1 xcolors=(node_colors.count(X)>0)?cn[X]:emptymap;
        get_nodes();
        inverse_ch();
        get_nodes();
        node_colors.erase(X);
        bleach(xcolors);
        get_nodes();
        upd_precendants();
        if ((descendants.count(start_node)==0)||(descendants[start_node].size()==0)) {descendants.clear(); ↙
    precendants.clear();}
        if ((precendants.count(top_node)==0)||(precendants[top_node].size()==0)) {descendants.clear();      ↙
    precendants.clear();}
    }
    void bleach(map1& xcolors) {
        map1::iterator it;
        map1 n0=nodes;
        for (it=n0.begin();it!=n0.end(); it++) {
            int node = (int)it->first;
            map1::iterator jt;
            dep_colors=xcolors;
            get_dep_colors(cn[node],xcolors,color_hierarchy,-1); // going down
            get_dep_colors(cn[node],xcolors,ich,1); // up
            map_diff(cn[node],dep_colors);
            remove_from_color_hierarchy(dep_colors);
            map_diff(node_colors[node],dep_colors);
            if (cn[node].size()==0) {
                descendants.erase(node);
                node_colors.erase(node);
                rmrefs(node,descendants);
                rmrefs(node,precendants);
                get_nodes();
                map1::iterator ci;
                for (ci=cn[node].begin();ci!=cn[node].end();ci++) { // for node we remove - deactivate it's ↙
     colors
                    int c=(int)ci->first;
                    color_nodes.erase(c);
                }
            }
        }
        clear_disconnected_colors();
    }
    void clear_disconnected_colors() {
        if (descendants.size()==0) return;
        inverse_ch();
        map1 rml;
        map2::iterator it;
        for (it=ich.begin(); it!=ich.end(); it++) {
```

C:\mydocs\deform_\sdg.h 5

```
            int c = (int)it->first;
            if ((ich[c].size()==0)&&(top_colors.count(c)==0)) {
                rml[c]=true;
            }
        }
        remove_from_color_hierarchy(rml);
        inverse_ch();
    }
    void remove_from_color_hierarchy(map1& x) {
        map1::iterator it;
        for (it=x.begin();it!=x.end();it++) {
            int c = (int)it->first;
            if (!active_color(c)) {
                color_hierarchy.erase(c);
                map1::iterator it;
                for (it=ich[c].begin();it!=ich[c].end();it++) {
                    int sc = (int)it->first;
                    color_hierarchy[sc].erase(c);
                }
                continue;
            }
            remove_one_color_from_color_hierarchy(c);
        }
    }
    void remove_one_color_from_color_hierarchy(int c) {
        // given color may have several ! sup colors: i.e. red is owned by blue and brown in different ↙
    halfs before merging together
        map1::iterator it;
        map1 x = color_hierarchy[c];
        color_hierarchy.erase(c);
        color_nodes.erase(c);
        if (ich[c].size()==1) {
            //color_hierarchy.erase(ich[c].begin()->first);
            ich.erase(c);
        }
        for (it=x.begin();it!=x.end();it++) {
            int sc = (int)it->first;
            if (!active_color(sc)) continue;
            if (sc==emptycolor) continue;
            if (ich[sc].size()<=1) {
                ich.erase(sc);
                remove_one_color_from_color_hierarchy(sc); // the other owner may be removed later, but ↙
    since this owner (c) was removed now we will finally delete sc
            }
        }
    }
    void extract_path() {
        path.clear();
        bool b=has_path_to_top(top_colors.begin()->first,emptymap,color_hierarchy,-1);
        colorpath=path;
        if (!b) {
            printf(" No path to top \n");
        }
        translate_to_nodes();
        int pathlen=path.size();
        if (pathlen==gmax_slack) {
            printf(" Path is confirmed \n");
        } else {
            printf(" Path is not confirmed \n");
        }
    }
    void translate_to_nodes() {
        map1 node_path;
        map1::iterator it;
        for (it=path.begin();it!=path.end();it++) {
            int c= (int)it->first;
```

```
            node_path[color_nodes[c]]=true;
            printf(" path %d %d \n",c,color_nodes[c]);
        }
        path=node_path;
    }
    map1 dep_colors;
    void get_dep_colors(map1& colors,map1& xcolors,map2& fiber,int dir) {
        // traverse to top for each color in colors avoiding xcolors, those that dont have path to top are ↙
    dep colors
        map1::iterator it;
        for (it=colors.begin(); it!=colors.end(); it++) {
            int c = (int)it->first;
            if ((c==emptycolor)&&(dir==-1)) continue;
            path.clear();visited.clear();
            if (!has_path_to_top(c,xcolors,fiber,dir)) dep_colors[c]=true;
        }
    }
    map1 path, colorpath, visited;
    bool has_path_to_top(int src,map1& xcolors,map2& ch,int dir) { // from src avoiding xcolors through ch
        map1::iterator it;
        if ((dir==1)&&(ch[src].size()==0)) return true;
        for (it=ch[src].begin();it!=ch[src].end();it++) {
            int c = (int)it->first;
            if (!active_color(c))  continue; // inactive color - every color is cn[] for some node, if that ↙
     node is removed color becomes inactive
            if (xcolors.count(c)>0) continue;
            if (path.size()>0) {
                if (dir==1) {
                    if (!map_contains(subcolors[c],path)) continue;
                } else {
                    if (!c_in_every_sub(c)) continue;
                }
            }
            path[c]=true;
            if ( ((ch[c].size()==0)&&(dir==1))||((c==emptycolor)&&(dir==-1)) ) return true;
            bool ret=has_path_to_top(c,xcolors,ch,dir);
            if (ret) return true;
        }
        path.erase(src);
        return false;
    }
    bool c_in_every_sub(int c) {
        map1::iterator it;
        for (it=path.begin();it!=path.end();it++) {
            int sc = (int)it->first;
            if (subcolors[sc].count(c)==0) {
                return false;
            }
        }
        return true;
    }
    int depth;
    map1 get_colored_nodes(int color) {
        map1 ret;
        map1::iterator it;
        for(it=nodes.begin();it!=nodes.end();it++) {
            int node = (int)it->first;
            if (node_colors[node].count(color)>0) ret[node]=true;
        }
        return ret;
    }
    map1 traversed;
    list<int> full_path;
    map1 traversed_color_nodes;
    map1 sup_colors;
    map1 emptymap;
```

C:\mydocs\deform_\sdg.h 7

```
};

struct base_edg : dg {
    static const int max_nodes=1000;
    bool **docked; // dock registry at the node
    base_edg() {
        docked = new bool*[max_nodes];
        for (int k=0; k<max_nodes; k++) {
            docked[k]=new bool[max_nodes];
            for (int j=0; j<max_nodes; j++) docked[k][j]=false;
        }
    }
    virtual bool add_delete(int node, int X)=0;
};
struct base_bot : util {
    int current_node;
    bool done;
    base_edg *p;
    base_bot(){};
    base_bot(int node, base_edg *p_) : current_node(node), p(p_){};
    virtual void advance()=0;
    virtual bool has_botlets()=0;
};
struct bot : public base_bot {
    map<int, bot> botlets; // newborn bots
    bot(){};
    bot(int node, base_edg *p_) : base_bot(node,p_) {
        done=false;
    };
    void advance() {
        botlets.clear();
        map1 &d=p->descendants[current_node];
        map1::iterator mit;
        int outbounds=0;
        done=true;
        _ASSERTE( _CrtCheckMemory( ) );
        int entry_node=current_node;
        for (mit=d.begin(); mit!=d.end(); mit++) {
            int node_out = mit->first;
            //if (visited.count(node_out)) continue;
            if (!p->add_delete(entry_node,node_out)) continue;
            if (!p->docked[node_out][step]) {
                p->docked[node_out][step]=true;
                if (outbounds>0) botlets[botlets.size()]=bot(node_out,p);
                else {
                    current_node=node_out;
                    done=false;
                }
                outbounds++;
            }
            //_ASSERTE( _CrtCheckMemory( ) );
            if ((node_out==end_node)&&(outbounds==1)) done=true;
        }
        if (outbounds==0) done=true;
    }
    bool has_botlets() {
        return (botlets.size()>0);
    }
};
struct slacks {
    int this_node;
    bool active;
    map<int,dg> *sgs; // dg per slack
    slacks() {};
    slacks& operator =(const slacks& s2) {
        this_node=s2.this_node;
```

C:\mydocs\deform_\sdg.h 8

```
        active=s2.active;
        sgs=new map<int,dg>();
        *sgs = *s2.sgs;
        return *this;
    }
    slacks(const slacks& s2) {
        this_node=s2.this_node;
        active=s2.active;
        sgs=new map<int,dg>();
        *sgs = *s2.sgs;
    }
    slacks(int this_node_) : this_node(this_node_){
        sgs=new map<int,dg>();
        active=true;
    };
    void add_slack(int new_node, int prev_node) {
        this_node=new_node;
        if (sgs->size()>0) {
            for (int slack=gmax_slack;slack>0;slack--) {
                if ((*sgs).count(slack)==0) continue;
                dg sdg = (*sgs)[slack];
                if (sdg.descendants.size()==0) {
                    sgs->erase(slack);
                    continue;
                }
                sdg.slack_level=slack+1;
                sdg.add(new_node,prev_node);
                (*sgs).erase(slack);
                sdg.get_nodes();
                int num_nodes=sdg.nodes.size();
                (*sgs)[sdg.slack_level]=sdg;
            }
        } else {
            if (prev_node==start_node) {
                dg sdg;
                sdg.add(new_node,prev_node);
                sdg.slack_level=1;
                (*sgs)[1]=sdg;
            }
        }
    }
    slacks& operator +=(slacks& s2) {
        // mtaching dgs are combined
        map<int,dg>::iterator mit;
        for (mit=sgs->begin(); mit!=sgs->end(); mit++) {
            int slack_level=(*mit).first;
            if (s2.sgs->count(slack_level)>0) {
                // merge the matching ones:
                (*sgs)[slack_level]+=(*s2.sgs)[slack_level];
            }
        }
        // new ones are added
        for (mit=s2.sgs->begin(); mit!=s2.sgs->end(); mit++) {
            int slack_level=(*mit).first;
            if (sgs->count(slack_level)==0) {
                (*sgs)[slack_level]=(*s2.sgs)[slack_level];
            }
        }
        return *this;
    }

    bool rm_node(int X,int Y) {
        map<int,dg>::iterator mit;
        list<int> rml;
        for (mit=sgs->begin(); mit!=sgs->end(); mit++) {
            dg &g=(*mit).second;
```

```
            int slack = (*mit).first;
            if ((g.precendants.count(X)==0)&&(g.descendants.count(X)>0)) {
                printf("WTF");
            }
            if (g.precendants.count(X)==0) continue; // one may bypass 4 others go through 4, still need to ↙
     process them !!!
            g.rm_node(X,Y);
            g.clear_inactive_colors();
            g.slack_level=(*mit).first;
            (*sgs)[(*mit).first]=g;
            if (g.descendants.size()==0) {
                rml.push_back((*mit).first);
                continue;
            }
        }
        list<int>::iterator it;
        for (it=rml.begin(); it!=rml.end(); it++) {
            sgs->erase((int)(*it));// lose that dag completely
        }
        _ASSERTE( _CrtCheckMemory( ) );
        return (rml.size()>0);
    }
};
struct edg : base_edg { // extended directed graph - slack graph is attached per node and slack level
    slacks *node_slacks; // int node mapped onto a slack map which stores sg for each slack
    edg() {hpath_cnt=2;}
    int slack_depth_check;
    int ei;
    map<int,bot> bots;
    int hpath_cnt;
    bool add_delete(int node, int X) {
        add_delete_cnt++;
        slacks ns(node_slacks[node]); // make a separate copy for each descendant
        if (ns.rm_node(X,node)) { // if this slacks stack sgs which contain node "it", they need to be      ↙
    pruned
            if (ns.sgs->size()==0) return false;
        }
        ns.add_slack(X,node);
        if (node_slacks[X].active) node_slacks[X]+=ns;
        else node_slacks[X]=ns;
        return true;
    }
    void bot_march() { // makes only one step at a time, returns control back
        map1 visited_;
        bots[0]=bot(start_node,this);
        key1=0;
        bool done=false;
        for(step=1;bots.size()>0;step++) {
            map<int,bot>::iterator it;
            map1 rmc;
            map<int,bot> new_bots;
            if (step>gmax_slack) {
                printf(" step>gmax_slack \n");
            }
            for (it=bots.begin(); it!=bots.end(); it++) {
                int key=it->first;
                it->second.advance();
                if (node_slacks[end_node].sgs->count(gmax_slack)>0) {
                    bots.clear();
                    printf(" march completed \n");
                    done=true;
                    break;
                }
                if (!it->second.done) {
                    if (it->second.has_botlets()) add_maps<map<int,bot>>(&new_bots,&it->second.botlets);
                } else rmc[key]=true;
```

C:\mydocs\deform_\sdg.h 10

```
            }
            if (done) break;
            map1::iterator mit;
            for (mit=rmc.begin(); mit!=rmc.end(); mit++) bots.erase(mit->first);
            add_maps<map<int,bot>>(&bots,&new_bots);
            printf(" march command %d, number bots %d \n",step,bots.size());
        }
    }
    void init_slacks() {
        node_slacks=new slacks[gmax_slack+2];
        new_color=1;
        map1::iterator mit2;
        map1 &nodes = get_nodes();
        //_ASSERTE( _CrtCheckMemory( ) );
        for (mit2=nodes.begin(); mit2!=nodes.end(); mit2++) {
            int X=(*mit2).first;
            node_slacks[X]=slacks(X);
        }
    }
    void upd_start_end() {
        // start node is one that has no precendants
        map2::iterator it;
        for (it=descendants.begin(); it!=descendants.end(); it++) {
            int node = it->first;
            if (precendants.count(node)==0) {
                start_node=node;
                break;
            }
        }
        for (it=precendants.begin(); it!=precendants.end(); it++) {
            int node = it->first;
            if (descendants.count(node)==0) {
                end_node=node;
                break;
            }
        }
    }
    void runit(int i) {
        subcolors.clear();
        ei=i;
        //_ASSERTE( _CrtCheckMemory( ) );
        load_graph(i);
        upd_start_end();
        init_slacks();
        bot_march();
        if ((*node_slacks[end_node].sgs).count(gmax_slack)>0) {
            printf(" H-Graph %d \n",i);
            (*node_slacks[end_node].sgs)[gmax_slack].extract_path();
        } else {
            printf(" Not an H-Graph %d \n",i);
        }
    }
};
```

C:\mydocs\deform_\util.h 1

```
#include <iostream>
#include <fstream>
#include <list>
#include <string>
#include <deque>
#include <list>
#include <map>
#include <time.h>

using namespace std;
#define smallnum -9999999
#define largenum 9999999

typedef map<int,bool> map1;
typedef map<int,map1> map2;
int gmax_slack;

#define isnan(x) ((x) != (x))
#define sign(x) (((x)>0)?1:(((x)==0)?0:-1))
#define round(x) (((x)-floor(x)>ceil(x)-(x))?ceil(x):floor(x))

static const char graphpath[] = "c:\\graphs\\";
struct util {
virtual void processTagValue(char* tag,char* value){}
void LoadConfig() {
  char* line=(char*)malloc(1024);
  ifstream fs;
  fs.open("config.txt",ios::in);
  if (fs.good()==0)     {
    printf("\n Error opening the config file, exiting.\n");
    throw "\n Error opening the config file, exiting.\n";
  }
  char* tag=(char*)malloc(256);
  char* value=(char*)malloc(256);
  for (int row=0;fs.good()!=0;row++)    {
    fs.getline(line, 1024, '\n');
    strcat(line,"\0");
    if (strstr(line,"=")==NULL) continue;
    strcpy(tag,strtok( line, "=" ));
    strcpy(value,strtok( NULL, "=" ));
    processTagValue(tag,value);
    ////_ASSERTE(_CrtCheckMemory());;;
  }
  fs.close();
}
static const int msz=102400;
int szl;

map2 descendants; // nodes onto children
map2 precendants; // for each child map onto parents

bool load_graph(int i) {
    // M = [0 1 0 1; 0 0 1 0; 1 0 0 0; 0 1 0 0] // matlab, adjacency matrix, generate // 1 means outbound ↙
    to somewhere, i.e. 1->2, 1->4
    // load descendants directly, precendants are obtained from descendants, where a gien node is a ↙
    descendant those are it's precendants
    char line[msz];
    szl=(int)round(sqrt((float)msz));
    ifstream fs;
    char fname[60];
    sprintf(fname,"%sgraph%d.txt",graphpath,i);
    fs.open(fname);
    if (fs.good()==0) {
      fs.close();
      return false;
```

```
    }
    fs.getline(line, msz, '\n');
    strcat(line,"\0");
    char * pch = strtok (line,";[]");
    int nn = msz/szl, k=0;
    char **node=new char*[nn+1];
    for (k=1; pch != NULL; k++) {
       pch = strtok (NULL, ";[]");
       if (pch==NULL) break;
       node[k]=new char[szl];
       strcpy(node[k],pch);
       strcat(node[k],"\0");
    }
    for (int j=1; j<k; j++) {
        char * pch = strtok (node[j]," ");
        for (int ni=1; pch != NULL; ni++) {
           int val = atoi(pch);
           if (val==1) {
               descendants[j][ni]=true;
               if (j==ni) {
                   printf(" j==ni \n");
               }
               if (descendants.count(ni)>0) {
                   if (descendants[ni].count(j)>0) {
                       printf(" circular reference \n");
                   }
               }
               precendants[ni][j]=true;
           }
           pch = strtok (NULL, " ");
           if (pch==NULL) break;
        }
    }
    gmax_slack=descendants.size();
    fs.close();
    return true;
}
map1 flatten(map2& a) {
    map1 ret;
    map2::iterator it;
    for (it=a.begin();it!=a.end();it++) {
        int key = (int)it->first;
        ret[key]=true;
        map1::iterator jt;
        for (jt=a[key].begin();jt!=a[key].end();jt++) {
            int v = (int)jt->first;
            ret[v]=true;
        }
    }
    return ret;
}
void gen_graph(int i) {
    int num_nodes=17,local_degree=2;
    while (true) {
        map2 rows;
        map<int,int> node_inds;
        map1 chk;
        bool restart=false;
        for (int k=1; k<num_nodes-1; k++) { // 1 and num_nodes are forbidden
            for(int s=0;true;s++) {
                int j=rand()%num_nodes;
                if (s>5*num_nodes) {
                    restart=true; // one last available number may match the index variable, restart then
                    break;
                }
                if ((chk.count(j)>0)||(j<2)||(j==k)) continue;
```

```
                node_inds[k]=j;
                chk[j]=true;
                break;
            }
            if (restart) break;
        }
        if (restart) continue;
        map<int,int> path;
        int node_ind=0;
        int prev_r=1;
        for (int r=1; r<num_nodes; r++) { // n-1 rows
            if (node_inds[r]==0) node_inds[r]=num_nodes;
            if (prev_r==node_inds[r]) {
                int i=0;
            }
            rows[prev_r][node_inds[r]]=true; // node r connects to node_ind
            if (path.count(prev_r)>0) {
                int i=0;
            }
            path[prev_r]=node_inds[r];
            prev_r=node_inds[r];
        }
        for (int r=1; r<num_nodes; r++) { // n-1 rows
            if ((r>1)&&(r<num_nodes-1)) { // one before last only has last one as descendant;
                int j=1;
                while (true) {
                    double d=rand()/((double)RAND_MAX);
                    int k=(int)(d*num_nodes);
                    if (k<2) continue; // 1 is start node, it's not anybodies descendant
                    if (k==num_nodes) continue; // num_nodes is end node, it's not anybodies descendant
                    if (rows[r].count(k)>0) continue;
                    if (rows[k].count(r)>0) continue; // at 6 we have 15 then at 15 we cant have 6
                    if (r==k) continue;
                    rows[r][k]=true;
                    j++;
                    if (j>local_degree) break;
                }
            }
        }
        save_graph(rows,i);
        char fname[128];
        sprintf(fname,"%shpath%d.txt",graphpath,i);
        save_map(path,fname);
        printf(" path length %d ",path.size());
        break;
    }
}
void save_graph(map2 &rows,int i){
    char fname[128];
    int num_nodes=rows.size()+1;
    sprintf(fname,"%sgraph%d.txt",graphpath,i);
    map2::iterator it;
    string str("M = [");
    int r=1;
    for (it=rows.begin(); it!=rows.end(); it++) {
        map1 &row=it->second;
        for (int k=1; k<=num_nodes; k++) {
            if (k>1) str+=" ";
            if (row.count(k)>0) {
                str+="1";
            } else str+="0";
        }
        r++;
        if (r!=rows.size()+1) str+="; ";
    }
    str+="]";
```

```
    ofstream ofs(fname,ios::out);
    ofs<< str << "\n";
    ofs.close();
}
void save_map(map<int,int> &a,char *fname){
    ofstream ofs(fname,ios::out);
    map<int,int>::iterator it;
    for (it=a.begin(); it!=a.end(); it++) {
        ofs<< it->first << " -> "<< it->second << "\n";
    }
    ofs.close();
}
  int get_cols(char *name) {
    char line[1024];
    ifstream fs;
    fs.open(name);
    if (fs.good()==0) {
      fs.close();
      return -1;
    }
    char *token;int col=0;
    fs.getline(line, 1024, '\n');
    strcat(line,"\0");
    token = strtok( line, "," );
    for(col=0; token != NULL; col++) {
      token = strtok( NULL, "," );
    }
    fs.close();
    return col;
  }
  int get_lines(const char *name) {
    char line[1024];
    ifstream fs;
    fs.open(name);
    if (fs.good()==0) {
      fs.close();
      return -1;
    }
    int row=0;
    for (row=0;fs.good()!=0;row++) {
      fs.getline(line, 1024, '\n');
      if (strlen(line)<1) return row;
    }
    fs.close();
    return row;
  }

  template<typename T>
  T* join_maps(T *a, T *b) {
      T::iterator mit;
      for (mit=b->begin(); mit!=b->end(); mit++) {
          int key=(*mit).first;
          if (a->count(key)==0) {
              (*a)[key]=(*b)[key];
          }
      }
      return a;
  }
  int key1;
  template<typename T>
  T* add_maps(T *a, T *b) {
      T::iterator mit;
      int j=a->size();
      for (mit=b->begin(); mit!=b->end(); mit++) {
          int key=(*mit).first;
          (*a)[key1]=(*b)[key];
```

C:\mydocs\deform_\util.h 5

```
        key1++;
    }
    return a;
}

void map_diff(map1 &a,map1 &b) { // need to avoid bleaching upstream from precendant of X, otherwise we ↙
  will always bleach all descendants of X
    map1::iterator it;
    for (it=b.begin(); it!=b.end(); it++) {
        int key=(*it).first;
        if (a.count(key)>0) a.erase(key);
    }
}
bool map_contains(map1 &a,map1 &b) { // map1 has map2
    map1::iterator it;
    for (it=b.begin(); it!=b.end(); it++) {
        int key=(*it).first;
        if (a.count(key)==0) return false;
    }
    return true;
}
void merge_maps(map2& a, map2& b) {
      map2::iterator mit;
      for (mit=a.begin(); mit!=a.end(); mit++) {
          int node=(*mit).first;
          if (b.count(node)>0) {
              join_maps(&a[node],&b[node]);
          }
      }
      // those that were missing in this graph just get added from g2:
      map2::iterator it;
      for (it=b.begin(); it!=b.end(); it++) {
          int node=(int)(*it).first;
          if (a.count(node)==0) {
              if (b.count(node)>0) a[node]=b[node];
          }
      }
}
void merge_maps(map<int,list<int>>& a, map<int,list<int>>& b) {
      map<int,list<int>>::iterator mit;
      // those that were missing in this graph just get added from g2:
      for (mit=b.begin(); mit!=b.end(); mit++) {
          int node=(int)(*mit).first;
          if (a.count(node)==0) {
              if (b.count(node)>0) a[node]=b[node];
          }
      }
}
void merge_maps(map<int,int>& a, map<int,int>& b) {// color_nodes
    map<int,int>::iterator it;
    for (it=b.begin();it!=b.end();it++) {
        int color=(int)it->first;
        a[color]=b[color];
    }
}
bool map_equals(map2& a, map2& b) {
    return a_in_b(a,b)&&a_in_b(b,a);
}
bool a_in_b(map2& a, map2& b) {
    map2::iterator it;
    for (it=a.begin();it!=a.end();it++) {
        int key = (int)it->first;
        if (b.count(key)==0) return false;
        map1::iterator jt;
        for (jt=a[key].begin();jt!=a[key].end();jt++) {
            int key2 = (int)jt->first;
```

```
            if (b[key].count(key2)==0) return false;
        }
        for (jt=b[key].begin();jt!=b[key].end();jt++) {
            int key2 = (int)jt->first;
            if (a[key].count(key2)==0) return false;
        }
    }
    return true;
  }
  map1 intersect(map1 &a, map1 &b) {
    // those in a that in b
    map1 ret;
    map1::iterator it;
    for (it=a.begin();it!=a.end();it++) {
        int key = (int)it->first;
        if (b.count(key)>0) {
            ret[key]=true;
        }
    }
    return ret;
  }
}; // util
```

C:\mydocs\deform_\deform.cpp 1

```
#include "stdafx.h"
#include "sdg.h"
void gen_graphs(int n) {
    edg gr;
    srand ( time(NULL) );
    for (int g=1; g<=n; g++) {
        printf(" generating graph %d \n",g);
        gr.gen_graph(g);
    }
}
int _tmain(int argc, _TCHAR* argv[]) {
    if (false) {
        gen_graphs(1000);
    } else {
        int gi=_ttoi(argv[1]);
        printf(" Processing Graph %d \n",gi);
        edg gr;
        gr.runit(gi);
    }
    return 0;
}
```